Brief Announcement: Hierarchical Consensus for Scalable Strong Consistency

Benjamin Bengfort
University of Maryland
College Park, Maryland 20742
bengfort@cs.umd.edu

Pete Keleher
University of Maryland
College Park, Maryland 20742
keleher@cs.umd.edu

ACM Reference format:
Benjamin Bengfort and Pete Keleher. 2017. Brief Announcement: Hierarchical Consensus for Scalable Strong Consistency. In *Proceedings of ACM Symposium on Principles of Distributed Computing, Washington, D.C., USA, July 2017 (PODC'17)*, 3 pages.
DOI: 10.475/123_4

1 INTRODUCTION

Strong consistency in a geo-replicated distributed data store requires a fault-tolerant mechanism that maintains consistency during node failure and communication partitions. Distributed consensus protocols inspired by Paxos [4] have been widely adopted to coordinate consistency, however, because of increased communication they cannot scale to arbitrary system sizes[2]. Several recent algorithms have attempted to address the scaling limitations of consensus and take two general forms — leader election and conflict detection.

Leader-oriented consensus such as MultiPaxos [4] and Raft [7] minimize the number of required communication phases by nominating a dedicated proposer. Less communication means better throughput, and fault-tolerance is achieved through node failure detection such as a heartbeat and a new leader is elected to minimize downtime. Leader-oriented approaches introduce two new problems, however: *load* and *distance*. Because the leader will necessarily do more work and handle more communication than other nodes, it must have sufficient resources to handle the workload; moreover, since any node can be elected leader, all nodes must have sufficient resources to handle the workload. This introduces scaling problems in two dimensions: adding nodes means more communication, increasing the minimum resource requirements for all nodes in the system. In geo-replicated systems, bandwidth and latency are highly variable therefore the election of a leader in a specific location means that consensus is bound by the slowest connection, making the consensus algorithm sensitive to distance. Although recent approaches such as S-Paxos [1] and Mencius [5] add load balancing to leader-oriented mechanisms, they cannot solve the distance problem.

Conflict detection approaches such as EPaxos [6] and MDCC [3] are optimistic that most consensus decisions are consistent. They propose "fast" and "slow" consensus paths, such that a subset of close nodes can quickly reach consensus but add dependency information to detect conflicts when commands are applied. If a conflict is detected, then decision must traverse the "slow" path to ensure correctness. Conflict detection does not have a distance problem, as nodes can select close neighbors, however this method does not guarantee dissemination of the command, which can require large amounts of dependency resolution. As the network scales, dependency graphs tend to increase in both size and complexity, increasing the load on the system.

In practice systems do not implement global consensus, but instead apply multiple consensus groups to coordinate specific objects or tablets. This keeps quorum sizes small, allocating just enough nodes to a quorum to maintain a minimum level of fault tolerance. However, in so doing, an object can only be consistent with respect to its own updates and the system loses information about dependencies. Moreover, there is no coordination between consensus groups, a single node can participate in multiple per-object consensus groups, which does not eliminate node and distance problems.

In this paper we introduce *Hierarchical Consensus*, an approach to generalize consensus that allows us to scale groups beyond a handful of nodes, across wide areas. Hierarchical Consensus (HC) increases the availability of consensus groups by partitioning the decision space and nominating a leader for each partition. Partitions eliminate distance by allowing decisions to be co-located with replicas that are responding to accesses. Hierarchical consensus is flexible locally, but provides strong global system guarantees.

2 CONSISTENCY AND CONSENSUS

We consider a set of processes $P = \{p_i\}_{i=1}^n$ which are connected via an asynchronous network, whose connections are highly variable. The variability of a communication link between p_i and p_j is modulated by the physical distance of the link across the geographic wide area. Each process maintains the state of a set of objects, $O = \{o_i^v\}_{i=1}^m$, which are accessed singly or in groups at a given process and whose state is represented by a monotonically increasing version number, v. There are two primary types of accesses, `read`, which returns the current versions of the accessed objects, and `update`, which increments the versions of the accessed objects.

Strong consistency across all process requires a global ordering of accesses. The strongest consistency, *linearizablity*, requires the ordering of both `read` and `update` operations. *Sequential consistency* allows for concurrent accesses so long as there is some globally defined ordering, which is specified by the `happens-before` relation ($\rightarrow$). Sequential consistency therefore only considers `update` operations but specifies an ordering of updates, maintaining a sequence $o_i^w \rightarrow o_i^{w+1} \rightarrow o_j^x$ and so forth. Objects that are accessed together

PODC'17, Washington, D.C., USA

DOI: 10.475/123_4

(by the same process and within a defined window of time) are implicitly dependent on each other, requiring their relative access order to be strictly defined. Objects that are not implicitly dependent on each other, e.g. accessed by separate processes, do not require a strict ordering and can instead be arbitrarily ordered by process id.

Distributed consensus protocols implement strong consistency by maintaining an ordered log of accesses (commands). Once a leader has been elected (a dedicated proposer), accesses are forwarded to the leader who, after checking application-specific invariants, broadcasts a request to other nodes to append the access to their log. A decision is reached when a majority of the quorum accept the append, at which point the leader sends a commit message. Fault-tolerance is observed by nodes that fall behind by replaying the log of committed accesses until they are up to date. Correctness is maintained by observing communications failure from the leader and electing a new leader.

3 HIERARCHICAL CONSENSUS

Hierarchical consensus implements coordination decisions as a tier of quorums such that parent quorums manage the decision space and leaf quorums manage access ordering. Hierarchical consensus considers the decision space as subsets of the object set, and subquorums are defined by a time-annotated disjoint subset of the objects they maintain, $Q_{i,e} \subset O$. The set of subquorums, Q is not a partition of O, but only represents the set of objects that are being accessed at time e.

The hierarchical consensus algorithm starts with a root quorum whose main responsibilities are i) the mapping of objects to subquorums and ii) the mapping of replicas to subquorums. Each instance of such a map defines a distinct *epoch*, E, a monotonically increasing representation of the term of Q_e. Decisions that require a modification of the decision space or changes the mapping of objects or replicas to subquorums requires a new *epoch*. The fundamental relationship between epochs is as follows: any access that happens in epoch $E_i \rightarrow E_{i+1}$. Alternatively, any access in epoch E_{i+1} *depends on* all accesses in epoch E_i. Accesses in different subquorums but in the same epoch happen concurrently from the global perspective, though accesses in a specific subquorm are totally ordered locally.

3.1 Operation

The parent consensus group must coordinate all decision space changes. Consider the simple example of the transfer of some objects from one subquorum to another:

$$Q_1 : \{ABC\} \rightarrowtail \{ABGH\}$$
$$Q_2 : \{DEFGH\} \rightarrowtail \{CDEF\}$$

Each of the two subquorums, Q_1 and Q_2, wants to give up a portion of its existing tag, and to add a portion of a tag currently mapped to another subquorum. We propose, initially, that this is a two phase process. Both subquorums make their requests to the leader, who may aggregate several namespace changes into a single one. While the root quorum gets consensus to make the epoch change, subquorums can continue operating on their own tags. Once the root quorum updates the epoch, it communicates the change to all affected subquorums. Each such subquorum then increments its epoch and acknowledges its change to the root quorum. At this point, the subquorum can operate on the portion of the tag it owned before, *but not the portion of the tag that is being modified.* Once the root quorum gets confirmations of epoch update from all subquorums, it notifies the subquorums that they can begin operating on the complete tag from the new epoch.

3.2 Epochs and Ordering

Let *interval* i_n be the ordered set of accesses of the replicas in subquorum Q_i during epoch E_n. We enforce sequentially consistent ordering of all accesses in the entire system by ensuring that there must exist a total ordering of the intervals that produces the correct access results. Access results should be equivalent to any interval ordering such that all intervals in E_n occur before intervals in E_{n+1} (our "interval ordering" invariant). This is because there is no cross-traffic between any Q_i and Q_j, and therefore ordering interval i_n before j_n is exactly the same as ordering interval j_n before i_n, for any i, j, and n.

3.3 Correctness

Only the subquorums that are involved in the tagspace changes need be notified and involved in the epoch change coordinated by the root epoch. Other subquorums can update their epoch number at no cost when they see the new epoch number from remote requests from other subquorums, or when they are notified by the root epoch. In this way, non-responding subquorums do not block tagspace changes for other quorums. Because the tag is left behind in the previous epoch, all writes in that tag will be ordered before writes in the next epoch. However, because no writes in the next epoch depend on these writes safety is still guaranteed.

This means that subquorums can have "fuzzy epochs", wherein some subquorums are behind others. Fuzzy epochs provide the flexibility needed to accommodate subquorums that may not be ready to move to a new epoch eliminate because application semantics (still accessing the same objects), or network conditions (in a tunnel).

We are investigating approaches to providing *eventual quiescence*: i.e. all tags eventually return to the root quorum if activity ceases. If implemented through either implicit or explicit leases, the root quorum would be empowered to pull a tag back when a lease expired. The ability for a tag to be pulled back could represent a loss of work performed by a partitioned subquorum, but is essential to the integrity of the system.

4 EXPERIMENTAL DESIGN

We propose to implement a distributed file system called FluidFS to more completely explore the use of hierarchical consensus in supporting file systems. FluidFS, implemented in the Go programming language, will allow us to quantitatively describe real-world environments and usage and to show how our proposed consistency model is experienced by users.

FluidFS will provide *close-to-open consistency* (CTO), meaning that a file open, which implies a full-file read, is guaranteed to see the data written by the "most recent" close of that file. Therefore file opens and closes must be totally ordered, and map easily onto operations in a replicated log. The canonical use of CTO is a single-server case, where a total ordering of file open and closes is just the ordering that the opens and closes arrive at the server. The distributed case assumed by this work is much more demanding because opens and closes are distributed across servers throughout the system.

We leverage hierarchical consistency to build a distributed set of sequentially-consistent logs that guarantee a total ordering over all file accesses. The result is a similar user experience to having the user/client co-located with a single server, while the user migrates around the system, using different devices, possibly collaborating with other users, and tolerating network vagaries, partitions, and failures.

Note that FluidFS, like many modern file systems, decouples metadata file data storage *recipes* Metadata includes an ordered list of *blobs*, which are opaque binary chunks. When a file is closed after editing, the data associated with the file is chunked into a series of variable-length blobs, identified by a hashing function applied to the data. The version created by the write access to the file specifies the blobs and their ordering that make up the file. Since blobs are effectively immutable, or tamper-evident, (blobs are named by hashes of their contents), we assert that consistent metadata replication can be decoupled from blob replication. Accesses to file system metadata becomes the operations or entries in the replicated logs. Metadata is therefore replicated through the system, allowing any file system client to have a complete view of the file system namespace.

5 DISCUSSION

Hierarchical consensus flexibly allocates subquorums to dynamic object groupings. Multiple subquorums means both more leaders and less global communication, reducing the resource requirements for most nodes, preventing bottlenecks, and increasing throughput. Consensus decisions can also be localized to where the accesses are occurring, minimizing both distance and the effect of wide area network variability. Finally, hierarchical consensus does not arbitrarily assign consensus decisions to single objects or unrelated groups of objects, but to objects that are implicitly dependent on each other because of their associated accesses.

An open question for our research is how to automatically allocate the namespace such that leadership of a subset of the namespace is local to the accesses and that members of the quorum are distributed to provide wide area durability and availability.

REFERENCES

[1] Martin Biely, Zoran Milosevic, Nuno Santos, and Andre Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 111–120. IEEE, 2012.
[2] H. Howard, D. Malkhi, and A. Spiegelman. Flexible Paxos: Quorum intersection revisited. *ArXiv e-prints*, August 2016.
[3] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM, 2013.
[4] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
[5] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for WANs. In *OSDI*, volume 8, pages 369–384, 2008.
[6] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372. ACM, 2013.
[7] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.